

Saying Hello World with GrGen.NET - A Solution to the TTC 2011 Instructive Case

Sebastian Buchwald Edgar Jakumeit
Karlsruhe Institute of Technology (KIT)
buchwald@kit.edu

We introduce the graph transformation tool GrGen.NET (www.grgen.net) by solving the Hello World Case [5] of the Transformation Tool Contest 2011 which consists of a collection of small transformation tasks; for each task a section is given explaining our implementation.

1 What is GrGen.NET?

GRGEN.NET is an application domain neutral graph rewrite system, the feature highlights regarding practical relevance are:

Fully Featured Meta Model: GRGEN.NET uses attributed and typed multigraphs with multiple inheritance on node/edge types. Attributes may be typed with one of several basic types, user defined enums, or generic set, map, and array types.

Expressive Rules, Fast Execution: The expressive and easy to learn rule specification language allows for a straightforward formulation of even complex problems, with an optimized implementation yielding high execution speed at modest memory consumption; outstanding features are iterated and recursive pattern matching and rewriting.

Programmed Rule Application: GRGEN.NET supports a high-level rule application control language, Graph Rewrite Sequences (GRS), offering sequential, logical, iterative and recursive control plus variables and storages for the communication of processing locations between rules.

Graphical Debugging: GRShell, GRGEN.NET's command line shell, offers interactive execution of rules, visualising together with yComp the current graph and the rewrite process. This way you can see what the graph looks like at a given step of a complex transformation and develop the next step accordingly. Or you can debug your rules and sequences.

2 Hello World!

The first task is to create a `Greeting` node with appropriate text. To solve the task we use the GRGEN rule from `HelloWorld.grg` shown below that creates the required graph when being executed:

```
rule createHelloWorld {
  replace {
    greeting:helloworld_Greeting;

    eval {
      greeting._text = "Hello World";
    }
  }
}
```

Rules in GrGen consist of a pattern part specifying the graph pattern to match and a nested rewrite part specifying the changes to be made. The pattern part is built up of node and edge declarations or references with an intuitive syntax: Nodes are declared by `n:t`, where `n` is an optional node identifier, and `t` its type. An edge `e` with source `x` and target `y` is declared by `x -e:t-> y`, whereas `-->` introduces an anonymous edge of type `Edge`. Nodes and edges are referenced outside their declaration by `n` and `-e->`, respectively. Attribute conditions can be given within `if`-clauses.

The rewrite part is specified by a `replace` or `modify` block nested within the rule. With `replace`-mode, graph elements which are referenced within the `replace`-block are kept, graph elements declared in the `replace`-block are created, and graph elements declared in the pattern, not referenced in the `replace`-part are deleted. With `modify`-mode, all graph elements are kept, unless they are specified to be deleted within a `delete()`-statement. Attribute recalculations can be given within an `eval`-statement. These and the language elements we introduce later on are described in more detail in the GrGen.NET user manual [1]; a discussion of the merits of textual compared to visual languages can be found in [4].

The first rule shown above consists of an empty pattern part and a `replace` part that creates a new node `greeting` of type `helloworld_Greeting` and evaluates the corresponding `_text` attribute. The creation rule for the extended metamodel given in Figure 3 is similar in structure, just more voluminous. A `Greeting` node is created and linked with a `greetingMessage` edge to a `GreetingMessage` node. Furthermore, it is linked with a `person` edge to a `Person` node. Then the attributes of the message and the person are initialized to the requested values (and the containment indices for XMI are set).

The files containing the rules seen so far start with a `using` statement `using helloworld__ecore;` which imports the GrGen metamodel which was generated from the given `Ecore` metamodel. GrGen.NET does not support importing `Ecore` metamodels directly (in contrast to GXL and native GRS files). Instead we supply an import filter generating an equivalent GrGen-specific graph model (`.gm` file) from a given `Ecore` file by mapping classes to GrGen node classes, their attributes to corresponding GrGen attributes, and their references to GrGen edge classes. Inheritance is transferred one-to-one, and enumerations are mapped to GrGen enums. Class names are prefixed by the names of the packages they are contained in to prevent name clashes; the same holds for references which are prefixed by their node class name, and node/edge attributes which are prefixed by an underscore. This name mangling can be seen in the first rule given, in the following listings it was removed due to space constraints and for the sake of readability, up to the migration chapter, from there on it is needed again to prevent type ambiguity. We regard this name mangling import of `Ecore` as the only weak point of the solution. Normally you are not interested in importing only a model, but you want to import an instance graph XMI (adhering to the `Ecore` model, thus adhering to the just generated equivalent GrGen graph model). Such an instance graph is imported by the filter, too, serving as the host graph for the following transformations (this can be seen in Figure 10, in this section we are only interested in the metamodel). The import process described is realized by the `import` command of the GrShell.

The GrShell offers the host environment for applying the rules given in the `.grg` files. The command line shell may be operated in interactive mode or in batch mode, for the tasks of this case we use it in batch mode by executing graph rewrite scripts, i.e. `.grs` files. The high level workflow of using GrGen is shown in Figure 1: the user written model (`.gm`) and rule (`.grg`) files are compiled by GrGen into `.NET` assemblies; together with a further runtime library these form the algorithmic core, to be accessed by user applications via the API offered by LibGr. One such application is the GrShell, accompanied by the graph viewer `yComp` it offers a rapid prototyping environment for graph transformation. For the first HelloWorld task the Shell executes the script given in Figure 4. The first line imports the `Ecore` metamodel `helloworld.ecore` and loads the transformation rules declared in the rule file `HelloWorld.grg`. The import process automatically generates the GRGEN metamodel file `helloworld__ecore.gm` that is used

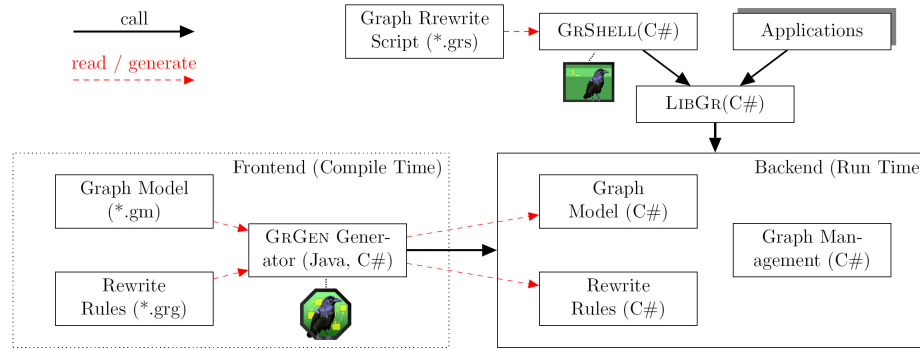


Figure 1: The structure of the GrGen.NET system

in the first rule and Figure 3. We then execute an extended **graph rewrite sequence** that consists of one application of the `createHelloWorld` rule. The graph rewrite sequences offer multiple operators for controlling rule execution and parameter passing between rules, several of them will be introduced later on, here we only execute a single parameterless rule once. The third line is used to show the resulting graph with the `yComp` tool, and finally the shell execution is quit.

Let us continue with the model-to-text transformation, Figure 5 shows the corresponding rule. The rule matches a node of type `Greeting` and the corresponding person and message (exactly what Figure 3 created). If a match is found, it creates a new node of type `StringResult` and assigns the concatenation of the text of the message and the name of the person to the `result` attribute. The `StringResult` is then emitted into an XMI file with the rule in Figure 6, which gets `#included` into the main rule file. The process is controlled by the shell script Figure 7. The 5th line is used to redirect the output of the `emit` statements from `stdout` to the specified file.

3 Count Matches with Certain Properties

The next task is to count the number of occurrences of certain graph structures. For each subtask the result needs to be wrapped in a node, this node is created by an application of the rule in Figure 8. The rule creates a node of type `IntResult`, initializes its `result` attribute to 0 and then returns the node out to the caller; it must be of type `IntResult` due to the output parameter declaration in the rule header with syntax `:(IntResult)`. (Alternatively we could create the node in the shell with the new command which is the preferred way for creating non-trivial initial host graphs.) The node with the count has to be written to an XMI file; this is accomplished with a text emitting rule `emitInt` nearly identical to the already introduced one `emitString` available in the file `Emitter.grg`, you may have a look at the GrGen.NET SHARE image [2] for details.

The first subtask consists of counting the number of nodes on the host graph. This is achieved by using the GRGEN rule shown in Figure 9. The rule increments the `result` attribute of the `IntResult` parameter by one. To get the count of all the nodes we execute the rule for all matches in the host graph—this can be requested in the graph rewrite sequences calling the rules by enclosing the rule name in all-brackets, as can be seen on line 4 of Figure 10. Having a closer look at this line we see that the subtask is handled by the successive application of 3 rules. The then-right operator `>` executes the left sequence and then the right sequence, returning as result of execution the result of the execution of the right sequence. The potential results of sequence execution are *success* equaling `true` and *failure*

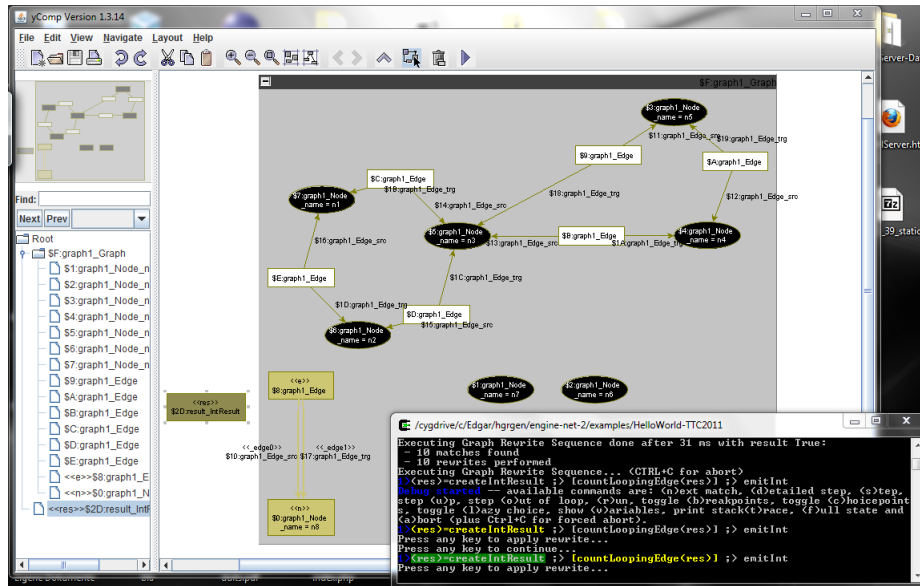


Figure 2: Debugging the sequence calling countLoopingEdge

equaling false; a rule which matches counts as success. The `IntResult` returned from the first rule is assigned to a variable `res`. This variable is read before executing the second rule, its value is handed in as input argument to the second rule, in fact to all applications of the second rule. The third rule emits (and deletes) the `IntResult` (it is not handed in, instead it gets matched in the `emitInt` rule). (Alternatively we could count the number of nodes of a certain type `T` with `show num nodes T` in the `GRSHELL`.)

The count looping edges subtask is interesting because in the shell script calling it the keyword `debug` was prepended before the `xgrs` command (cf. line 6 of Figure 10). This causes sequence execution to start in debug mode, i.e. yComp is started visualizing the host graph and the rule matches of interest, and the sequence is executed stepwise under user control. The situation right after the one match available for `countLoopingEdge` was found is displayed in Figure 2 below, the graph elements are annotated with the names of the pattern elements which matched them, cf. Figure 11.

A major feature of yComp is its high configurability. In line 2 of Figure 10 we include a further shell script given in Figure 12, which is used to achieve the nice layout displayed in Figure 2. You can use one of several available layout algorithms—with hierarchic and organic being the most useful ones, here we use circular. You can configure for every available node or edge type in which color with what node shape or edge style it should be shown, with what attribute values or fixed text as element labels or tags it is to be displayed, or if it should be shown at all. We used it to distinguish the Node nodes from the Edge nodes. Furthermore you can configure graph nesting by registering edges at certain nodes to define a containment hierarchy, causing the nodes to become displayed as subgraphs containing the elements to which they are linked by the given edges. This can be seen on lines 2 and 3 of Figure 12 which cause all Node nodes and all Edge nodes to be contained in the graph node.

Figure 13 shows the rule for the optional subtask of matching all dangling edges. It matches an Edge node and then uses an alternative to match either the `missingSrc` pattern or the `missingTrg` pattern. The `missingTrg` pattern matches the edge to the source node and uses a negative application condition (NAC) to ensure that the graph edge has no target node. A negative pattern causes the matching of the enclosing pattern to fail if it is found in the graph. Likewise, the `missingSrc` only matches an graph

edge with a target node, but without a source node. The rule `countIsolatedNode` in Figure 14 is not matching as soon as one of the negatives is found. The `countCycle` in Figure 15 is a direct encoding of the specification, so there is no need to go into depth here.

4 Reverse Edges

To solve this task we need to reverse all edges. The GRGEN rule from `Reverse.grg` shown below employs retyping also known as relabeling on the GRGEN edges to accomplish this task:

```
rule reverseEdge {
  alternative {
    Src {
      -src:src->;
      modify { -:trg<src>->; }
    }
    Trg {
      -trg:trg->;
      modify { -:src<trg>->; }
    }
  }
}
```

Retyping is specified with the syntax `y:t<x>`: this defines `y` to be a retyped version of the original node `x`, retyped to the new type `t`; for edges the syntax is `-y:t<x>->`. After applying the rule with `xgrs [reverseEdge]` all source nodes are target nodes and all target nodes are source nodes. This approach naturally reverses even dangling edges.

5 Simple Migration

To solve this task we need to migrate the graph from the graph metamodel used in the previous tasks to another graph metamodel which is characterized by introducing a superclass `GraphComponent` for `Node` nodes and `Edge` nodes. Since the target metamodel has a similar structure, we simply use retyping as introduced in the previous section to migrate the graph, with the rules given in Figure 16 controlled by the sequence given in Figure 17. To keep things simple we offer an endogenous transformation, an exogenous one would be possible as well by matching all nodes, creating their counterparts, and writing traceability information to a storagemap, then matching all edges, creating their counterparts by looking up the correct nodes from the storagemap. The name mangling from `Ecore` import is kept here (from now on) as removing it would render the types ambiguous.

In Figure 17 we apply each rule as long as possible (in each successive step searching one match then rewriting it, in contrast to the all bracketing introduced before which first collects all matches and then rewrites them at once). This is denoted by the postfix star `*` causing the preceding sequence to be iterated as long as it succeeds. The result of a star iteration is always success (in contrast to the plus `+` postfix which requires the preceding sequence to match at least once in order to succeed), so the complete sequence linked by strict conjunction operators `&` succeeds (`true`), too. Disjunction `|` is available as well, so are the lazy versions `&&` and `||` of the operators not executing the right sequence in case the result of the left sequence already determines the outcome.

The solution for the second (optional) target metamodel is similar to the first, so we only highlight the key difference: the second metamodel realizes edges by edges and not nodes anymore. Thus model

migration requires a non-isomorphic transformation step for edges, here we use the rule shown in Figure 18 and an additional rule to delete dangling edges. Striving for perfection, we order the `linksTo` references by migrating the `graphEdge.index` and employing an additional fix-up rule that ensures that the index falls into the interval $0 \dots |\text{linksTo}| - 1$. Thus the (outgoing) edges are ordered the same way as they are ordered in the original graph.

6 Delete Node with Specific Name and its Incident Edges

Deleting a node with a given name is a trivial task, which can be seen in the rule given in Figure 19 using `modify` mode explicitly deleting the matched node if it bears the name searched for. The (optional) subtask of also deleting all incident edges is more interesting, the rule from `Delete.grg` given below shows how this can be accomplished in GRGEN:

```
rule deleteN1AndAllIncidentEdges {
  n:graph1_Node;
  if {n._name == "n1";}

  iterated {
    n <-- e:graph1_Edge;

    replace { }
  }
}
```

The rule above matches the node with the name `n1` and all incident edges in an `iterated` fashion. The `iterated` construct munches eagerly the contained pattern as long as it is available in the graph and not yet matched; it succeeds even if the contained pattern is not available in the graph, in contrast to the similar `multiple` construct which requires the pattern to be available at least once causing matching of the enclosing pattern to fail otherwise. Since the `replace` parts are empty, all matched elements are deleted. If edges were real edges and not nodes we could just delete the node, due to SPO-semantics the edges would be removed too.

7 Insert Transitive Edges

The last (optional) task is to compute $R \cup R^2$ from a graph representation of a relation R . To solve this task we provide a rule `insertTransitiveEdge` given in Figure 20 that inserts a transitive edge if it is not available yet; we apply it on all matches by `xgrs [insertTransitiveEdge]`. The edges can not be given directly in the pattern but must be enclosed in the positive application condition patterns denoted by the `independents` in order to ensure that even for a multigraph with multiple edges between two nodes the transitive edges are not inserted multiple times. The `hom(n1,n2,n3)` statement allows a homomorphic matching for the contained nodes, i.e. they *can* be matched to the same host graph node. As a side remark: computing R^* would require nothing more than `xgrs insertTransitiveEdge*` (then we could even move the edges out of the `independents`; this was already the case in the first version of our solution for $R \cup R^2$ with the side effect of rendering it overly complicated, thanks to Gábor Bergmann for his hint to just employ an existentially quantified pattern, i.e. `independent`). One could even compute the transitive closure only on demand in graph queries which require it by employing recursive patterns, similar to how they are used in our Program Understanding case solution [3].

8 Conclusion

In this paper we presented a GrGen.NET solution to the Hello World! challenge of the Transformation Tool Contest 2011. We were able to solve all tasks of the challenge, including the optional tasks, introducing the reader alongside to a respectable amount of the functionality of GrGen.NET.

References

- [1] Jakob Blomer, Rubino Geiß & Edgar Jakumeit (2011): *The GrGen.NET User Manual*. <http://www.grgen.net>.
- [2] Edgar Jakumeit & Sebastian Buchwald (2011): *SHARE demo related to the paper Saying Hello World with GrGen.NET - A Solution to the TTC 2011 Instructive Case*. http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_GrGen_v2.vdi.
- [3] Edgar Jakumeit & Sebastian Buchwald (2011): *Solving the TTC 2011 Reengineering Case with GrGen.NET*. In Van Gorp et al. [6].
- [4] Edgar Jakumeit, Sebastian Buchwald & Moritz Kroll (2010): *GrGen.NET*. *International Journal on Software Tools for Technology Transfer (STTT)* 12, pp. 263–271, doi:10.1007/s10009-010-0148-8.
- [5] Steffen Mazanek (2011): *Hello World! An Instructive Case for the Transformation Tool Contest*. In Van Gorp et al. [6].
- [6] Pieter Van Gorp, Steffen Mazanek & Louis Rose, editors (2011): *TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011, Post-Proceedings*. EPTCS.

A Code Listings

```
rule createHelloWorldExt {
  replace {
    greeting:Greeting;
    greeting -e:greetingMessage-> message:GreetingMessage;
    greeting -f:person-> person:Person;

    eval {
      message.text = "Hello";
      person.name = "TTC□Participants";

      e.index = 0;
      f.index = 0;
    }
  }
}
```

Figure 3: HelloWorldExt.grg

```
import helloworld.ecore HelloWorld.grg
xgrs createHelloWorld
show graph ycomp
quit
```

Figure 4: HelloWorld.grs

```
#include "Emitter.gri"

rule messageToText {
  greeting:Greeting;
  greeting -:greetingMessage-> message:GreetingMessage;
  greeting -:person-> person:Person;

  replace {
    result:StringResult;

    eval {
      result.result = message.text + " " + person.name + "!";
    }
  }
}
```

Figure 5: HelloWorldExt.grg

```
rule emitString {
  string:StringResult;

  replace {
    emit("<?xml encoding=\"1.0\" encoding=\"ASCII\"?>\n");
    // some further lines emitting XMI text not displayed
    emit("result=\n" + string.result + "\n/>\n");
  }
}
```

Figure 6: Emitter.gri

```
import helloworldext.ecore result.ecore HelloWorldExt.grg
xgrs createHelloWorldExt
show graph ycomp
xgrs messageToText
redirect emit "Model-to-text-grgen.xmi"
xgrs emitString
quit
```

Figure 7: HelloWorldExt.grs


```

rule createIntResult : (IntResult) {
    modify {
        res: IntResult;

        eval {
            res.result = 0;
        }

        return (res);
    }
}

```

Figure 8: Count.grg

```

rule countNode(res: IntResult) {
    n: Node;

    modify {
        eval {
            res.result = res.result + 1;
        }
    }
}

```

Figure 9: Count.grg

```

import graph1.ecore result.ecore Graph1.xmi Count.grg
include layout.grsi
redirect emit "Count-nodes-grgen.xmi"
xgrs (res)=createIntResult ;> [countNode(res)] ;> emitInt
redirect emit "Count-looping-edges-grgen.xmi"
debug xgrs (res)=createIntResult ;> [countLoopingEdge(res)] ;> emitInt
# the other subtasks follow the two-line scheme given above
quit

```

Figure 10: Count.grs

```

rule countLoopingEdge(res: IntResult) {
    e: Edge - :src-> n: Node;
    e      - :trg-> n;

    modify {
        eval {
            res.result = res.result + 1;
        }
    }
}

```

Figure 11: Count.grg

```

debug set layout Circular
dump add node Graph group by hidden outgoing edges
dump add node Graph group by hidden outgoing nodes
dump set node Graph color lightgrey
dump set node Node shape circle
dump set node Node color black
dump set node Node textcolor white
dump add node Node infotag name
dump set node Edge color white

```

Figure 12: layout.grsi

```

rule countDanglingEdge(res:IntResult) {
  e:Edge;

  alternative {
    missingTrg {
      e -:src->;
      negative {
        e -:trg->;
      }

      modify { }
    }
    missingSrc {
      e -:trg->;
      negative {
        e -:src->;
      }

      modify { }
    }
  }

  modify {
    eval {
      res.result = res.result + 1;
    }
  }
}

```

Figure 13: Count.grg

```
rule countIsolatedNode(res:IntResult) {  
  n:Node;  
  negative {  
    n < -:src-;  
  }  
  negative {  
    n < -:trg-;  
  }  
  
  modify {  
    eval {  
      res.result = res.result + 1;  
    }  
  }  
}
```

Figure 14: Count.grg

```
rule countCycle(res:IntResult) {  
  n1:Node < -:src- :Edge -:trg-> n2;  
  n2:Node < -:src- :Edge -:trg-> n3;  
  n3:Node < -:src- :Edge -:trg-> n1;  
  
  modify {  
    eval {  
      res.result = res.result + 1;  
    }  
  }  
}
```

Figure 15: Count.grg

```

using graph1__ecore, graph2__ecore;

rule migrateGraph {
    n:graph1_Graph;
    modify {
        :graph2_Graph<n>;
    }
}

rule migrateNode {
    n:graph1_Node;
    modify {
        migrated:graph2_Node<n>;
        eval { migrated._text = n._name; }
    }
}

rule migrateEdge {
    e:graph1_Edge;
    modify {
        migrated:graph2_Edge<e>;
        eval { migrated._text = ""; }
    }
}

rule migrateEdgeSrc {
    -e:graph1_Edge_src->;
    modify {
        -migrated:graph2_Edge_src<e>->;
        eval { migrated.index = e.index; }
    }
}

rule migrateEdgeTrg {
    -e:graph1_Edge_trg->;
    modify {
        -migrated:graph2_Edge_trg<e>->;
        eval { migrated.index = e.index; }
    }
}

rule migrateGraphEdges {
    -e:graph1_Graph_edges->;
    modify {
        -migrated:graph2_Graph_gcs<e>->;
        eval { migrated.index = e.index; }
    }
}

rule migrateGraphNode {
    -e:graph1_Graph_nodes->;
    modify {
        -migrated:graph2_Graph_gcs<e>->;
        eval { migrated.index = e.index; }
    }
}

```

Figure 16: SimpleToEvolved.grg

```
xgrs migrateGraph* & migrateNode* & migrateEdge* & migrateEdgeSrc*\
    & migrateEdgeTrg* & migrateGraphEdges* & migrateGraphNodes*
```

Figure 17: SimpleToEvolved.grs

```
rule migrateEdge {
  e:graph1_Edge;
  e <-graphEdge:graph1_Graph_edges-;
  e -:graph1_Edge_src-> src:Node;
  e -:graph1_Edge_trg-> trg:Node;

  modify {
    delete(e);

    src -migrated:graph3_Node_linksTo-> trg;

    eval {
      migrated.index = graphEdge.index;
    }
  }
}
```

Figure 18: SimpleToMoreEvolved.grg

```
rule deleteN1 {
  n:graph1_Node;

  if {n._name == "n1";}

  modify {
    delete(n);
  }
}
```

Figure 19: Delete.grg

```
rule insertTransitiveEdge {
  n1:graph1_Node;
  n3:graph1_Node;
  hom(n1,n3);

  independent {
    hom(n1,n2,n3);
    n1 <=:graph1_Edge_src- :graph1_Edge -:graph1_Edge_trg-> n2:graph1_Node;
    n2 <=:graph1_Edge_src- :graph1_Edge -:graph1_Edge_trg-> n3;
  }
  negative {
    n1 <=:graph1_Edge_src- :graph1_Edge -:graph1_Edge_trg-> n3;
  }

  modify {
    n1 <=:graph1_Edge_src- :graph1_Edge -:graph1_Edge_trg-> n3;
  }
}
```

Figure 20: TransitiveSimple.grg